

HW6 : lab2 실습

20160394 임효상

※ 설명을 위해 부득이하게 사진을 더 추가, 페이지 수가 약간 더 늘어났습니다.

1) 실습 A

The screenshot displays a debugger interface with two main panels. The left panel shows assembly code for a function named `example1`. The right panel shows the memory and registers view.

Assembly Code (Left Panel):

```
example1();  
    0x1052c blt0x1053c <example1>  
    0x10530 movtr3, #0  
    0x10534 movtr0, r3  
    0x10538 poptr{r11, pc}  
  
return 0;  
    0x1053c pusht{r11, lr}  
    0x10540 addtr11, sp, #4  
    0x10544 subtrsp, sp, #10  
    0x10548 ldtr3, [pc, #70]!; 0x1059c <...>  
    0x1054c ldtr3, [r3]  
    0x10550 strtr3, [r11, #-8]  
    0x10554 movtr3, #0  
  
int b;  
int *ptr;  
  
ptr = &b;  
    0x10558 subtr3, r11, #10  
    0x1055c strtr3, [r11, #-12]  
  
ptr = 10;  
    0x10560 ldtr3, [r11, #-12]  
    0x10564 movtr2, #10  
    0x10568 strtr2, [r3]  
  
b = 250;  
    0x1056c movtr3, #250!; 0x100  
    0x10570 strtr3, [r11, #-16]  
  
int b;  
int *ptr1;  
  
ptr1 = &b;  
    0x10574 poptrt; (mov r0, r0)  
    0x10578 ldtr2, [pc, #28]!; 0x1059c <...>  
    0x1057c ldtr2, [r3]  
    0x10580 ldtr3, [r11, #-8]  
    0x10584 corstr2, r3, r2  
    0x10588 movtr3, #0  
    0x1058c beq0x10594 <example1+88>  
    0x10590 blt0x305dc <_stack_chk_fail...>  
    0x10594 subtrsp, r11, #4  
    0x10598 poptr{r11, pc}  
    0x1059c tt; <UNDEFINED> instruction: 0
```

Memory View (Right Panel):

address	hex	char
0xfffff13c	0x f1 fe ff	X...
0xfffff140	3c f1 fe ff
0xfffff144	20 9b 00 00
0xfffff148	54 f1 fe ff	T...
0xfffff14c	30 05 01 00	0...
0xfffff150	00 00 00 00
0xfffff154	e4 09 01 00
0xfffff158	00 00 00 00

Registers View (Right Panel):

name	value (hex)	value (decimal)	descriptor
r0	0x1	1	
r1	0xfffff294	4294898324	
r2	0xfffff29c	4294898332	
r3	0xfffff13c	4294897980	
r4	0x10ebc	69308	
r5	0x10158	65880	
r6	0xb408	570376	
r7	0x10158	65880	
r8	0x0	0	register 8 (64-bit)

*ptr = 16 명령어를 수행하기 직전의 모습이다.

r11에는 sp+4의 주소가 들어있다.

SUB r3, r11, #16 명령어로 인해 r3에는 a의 데이터가 저장된 메모리의 주소(r11-16)가 들어있다.

STR r3, [r11, #-12]명령어로 인해 stack의 r11-12번지 주소에는 b의 주솟값이 저장된다.

```

void example1()
{
    // ...
    int b;
    int *ptr;

    ptr = &b;

    *ptr = 10;

    b = 250;

    // ...
}

```

```

0x1053c pushr11, lr
0x10540 addr11, sp, #4
0x10544 subtsr, sp, #10
0x10548 ldrtr3, [pc, #70]t; 0x1059c <
0x1054c ldrtr3, [r3]
0x10550 strtr3, [r11, #-8]
0x10554 movtr3, #0

0x10558 subtr3, r11, #10
0x1055c strtr3, [r11, #-12]

0x10560 ldrtr3, [r11, #-12]
0x10564 movtr2, #10
0x10568 strtr2, [r3]

0x1056c movtr3, #250t; 0x100
0x10570 strtr3, [r11, #-10]

0x10574 noptrt; (mov r0, r0)
0x10578 ldrtr3, [pc, #28]t; 0x1059c <
0x1057c ldrtr2, [r3]
0x10580 ldrtr3, [r11, #-8]
0x10584 eorstr2, r3, r2
0x10588 movtr3, #0
0x1058c beqr0x10594 <example1+88>
0x10590 bltr0x305dc <_stack_chk_fail_
0x10594 subtsr, r11, #4
0x10598 poptr(r11, pc)
0x1059c tt; <UNDEFINED> instruction: 0

```

address	hex	char
more		
0xfffff13c	58 01 01 00	X...
0xfffff140	3c f1 fe ff
0xfffff144	20 9b 08 00
0xfffff148	54 f1 fe ff	T...
0xfffff14c	30 05 01 00	0...
0xfffff150	00 00 00 00
0xfffff154	e4 09 01 00
0xfffff158	00 00 00 00
more		

registers			
name	value (hex)	value (decimal)	description
r0	0x1	1	
r1	0xfffff294	4294898324	
r2	0xfffff29c	4294898332	
r3	0xfffff13c	4294897980	
r4	0x10ebc	69308	
r5	0x10158	65880	
r6	0xb408	570376	
r7	0x10158	65880	
r8	0x0	0	register 8 (64-bit)

LDR r3, [r11, #-12] 명령어 수행 : r11-12 메모리 주소에는 b의 주솟값(r11-16)이 들어있다. 이 값을 r3에 저장한다. 하지만 기존 r3에 이미 b의 주솟값이 들어있으므로 변화는 없다.

```

void example1()
{
    // ...
    int b;
    int *ptr;

    ptr = &b;

    *ptr = 10;

    b = 250;

    // ...
}

```

```

0x1053c pushr11, lr
0x10540 addr11, sp, #4
0x10544 subtsr, sp, #10
0x10548 ldrtr3, [pc, #70]t; 0x1059c <
0x1054c ldrtr3, [r3]
0x10550 strtr3, [r11, #-8]
0x10554 movtr3, #0

0x10558 subtr3, r11, #10
0x1055c strtr3, [r11, #-12]

0x10560 ldrtr3, [r11, #-12]
0x10564 movtr2, #10
0x10568 strtr2, [r3]

0x1056c movtr3, #250t; 0x100
0x10570 strtr3, [r11, #-10]

0x10574 noptrt; (mov r0, r0)
0x10578 ldrtr3, [pc, #28]t; 0x1059c <
0x1057c ldrtr2, [r3]
0x10580 ldrtr3, [r11, #-8]
0x10584 eorstr2, r3, r2
0x10588 movtr3, #0
0x1058c beqr0x10594 <example1+88>
0x10590 bltr0x305dc <_stack_chk_fail_
0x10594 subtsr, r11, #4
0x10598 poptr(r11, pc)
0x1059c tt; <UNDEFINED> instruction: 0

```

address	hex	char
more		
0xfffff13c	58 01 01 00	X...
0xfffff140	3c f1 fe ff
0xfffff144	20 9b 08 00
0xfffff148	54 f1 fe ff	T...
0xfffff14c	30 05 01 00	0...
0xfffff150	00 00 00 00
0xfffff154	e4 09 01 00
0xfffff158	00 00 00 00
more		

registers			
name	value (hex)	value (decimal)	description
r0	0x1	1	
r1	0xfffff294	4294898324	
r2	0x10	16	
r3	0xfffff13c	4294897980	
r4	0x10ebc	69308	
r5	0x10158	65880	
r6	0xb408	570376	
r7	0x10158	65880	
r8	0x0	0	register 8 (64-bit)

MOV r2, #9 : r2에 16의 값을 저장했다.

```

void example1()
{
    // ...
    int b;
    int *ptr;

    ptr = &b;

    *ptr = 10;

    b = 256;

    // ...
}

```

```

0x1053c push{r11, lr}
0x10540 add{r11, sp, #4}
0x10544 sub{sp, #10}
0x10548 ldtr{r3, [pc, #70]}; 0x1053c <=
0x1054c ldtr{r3, [r3]}
0x10550 strtr{r11, #-8}
0x10554 movtr{r3, #0}

0x10558 subtr{r11, #10}
0x1055c strtr{r11, #-12}

0x10560 ldtr{r3, [r11, #-12]}
0x10564 movtr{r2, #10}
0x10568 strtr{r2, [r3]}

0x1056c movtr{r3, #256}; 0x100
0x10570 strtr{r11, #-10}

0x10574 nop{tt}; (mov r0, r0)
0x10578 ldtr{r3, [pc, #28]}; 0x1053c <=
0x1057c ldtr{r2, [r3]}
0x10580 ldtr{r3, [r11, #-8]}
0x10584 cortr{r3, r2}
0x10588 movtr{r3, #0}
0x1058c beq{0x10594 <example1+88>}
0x10590 bit{0x305dc <__stack_chk_fail_1}
0x10594 sub{sp, #4}
0x10598 pop{r11, pc}
0x1059c tt; <UNDEFINED> instruction: 0

```

address	hex	char
more		
0xfffff13c	10 00 00 00
0xfffff140	3c f1 fe ff
0xfffff144	20 9b 08 00
0xfffff148	54 f1 fe ff	T...
0xfffff14c	30 05 01 00	0...
0xfffff150	00 00 00 00
0xfffff154	e4 09 01 00
0xfffff158	00 00 00 00
more		

breakpoints
signals
registers

name	value (hex)	value (decimal)	description
r0	0x1	1	
r1	0xfffff294	4294898324	
r2	0x10	16	
r3	0xfffff13c	4294897980	
r4	0x10ebc	69308	
r5	0x10158	65880	
r6	0x8b488	570376	
r7	0x10158	65880	
r8	0x0	0	register 8 (64-bit)

STR r2, [r3] : r3에 저장된 값인 b의 주솟값(r11-16)에 r2의 값(16)을 저장했다. 따라서 b의 값은 16다.

```

void example1()
{
    // ...
    int b;
    int *ptr;

    ptr = &b;

    *ptr = 10;

    b = 256;

    // ...
}

```

```

0x1053c push{r11, lr}
0x10540 add{r11, sp, #4}
0x10544 sub{sp, #10}
0x10548 ldtr{r3, [pc, #70]}; 0x1053c <=
0x1054c ldtr{r3, [r3]}
0x10550 strtr{r11, #-8}
0x10554 movtr{r3, #0}

0x10558 subtr{r11, #10}
0x1055c strtr{r11, #-12}

0x10560 ldtr{r3, [r11, #-12]}
0x10564 movtr{r2, #10}
0x10568 strtr{r2, [r3]}

0x1056c movtr{r3, #256}; 0x100
0x10570 strtr{r11, #-10}

0x10574 nop{tt}; (mov r0, r0)
0x10578 ldtr{r3, [pc, #28]}; 0x1053c <=
0x1057c ldtr{r2, [r3]}
0x10580 ldtr{r3, [r11, #-8]}
0x10584 cortr{r3, r2}
0x10588 movtr{r3, #0}
0x1058c beq{0x10594 <example1+88>}
0x10590 bit{0x305dc <__stack_chk_fail_1}
0x10594 sub{sp, #4}
0x10598 pop{r11, pc}
0x1059c tt; <UNDEFINED> instruction: 0

```

address	hex	char
more		
0xfffff13c	10 00 00 00
0xfffff140	3c f1 fe ff
0xfffff144	20 9b 08 00
0xfffff148	54 f1 fe ff	T...
0xfffff14c	30 05 01 00	0...
0xfffff150	00 00 00 00
0xfffff154	e4 09 01 00
0xfffff158	00 00 00 00
more		

breakpoints
signals
registers

name	value (hex)	value (decimal)	description
r0	0x1	1	
r1	0xfffff294	4294898324	
r2	0x10	16	
r3	0x100	256	
r4	0x10ebc	69308	
r5	0x10158	65880	
r6	0x8b488	570376	
r7	0x10158	65880	
r8	0x0	0	register 8 (64-bit)

MOV r3, #3 : r3에 256의 값을 저장했다.

```

void example1()
{
    int b;
    int *ptr;

    ptr = &b;

    *ptr = 10;

    b = 256;

    int b;
    int *ptr1;

    ptr1 = &b;
    *ptr1 = 10;

    b = 256;
}

```

```

0x1053c push{r11, lr}
0x10540 add{r11, sp, #4}
0x10544 sub{sp, sp, #10}
0x10548 ldr{r3, [pc, #70]}; 0x1053c <...
0x1054c ldr{r3, [r3]}
0x10550 str{r3, [r11, #8]}
0x10554 mov{r3, #0}

0x10558 sub{r3, r11, #10}
0x1055c str{r3, [r11, #-12]}

0x10560 ldr{r3, [r11, #-12]}
0x10564 mov{r2, #10}
0x10568 str{r2, [r3]}

0x1056c mov{r3, #256}; 0x100
0x10570 str{r3, [r11, #-10]}

0x10574 nop{tt}; (mov r0, r0)
0x10578 ldr{r3, [pc, #28]}; 0x1059c <...
0x1057c ldr{r2, [r3]}
0x10580 ldr{r3, [r11, #-8]}
0x10584 eor{r2, r3, r2}
0x10588 mov{r3, #0}
0x1058c beq{0x10594, <example1+88>}
0x10590 blt{0x1059c, <_stack_chk_fail...
0x10594 sub{sp, r11, #4}
0x10598 pop{r11, pc}
0x1059c tt; <UNDEFINED> instruction: 0

```

address	hex	char
move		
0xfffff13c	00 01 00 00
0xfffff140	3c f1 fe ff
0xfffff144	20 9b 00 00
0xfffff148	54 f1 fe ff	T...
0xfffff14c	30 05 01 00	0...
0xfffff150	00 00 00 00
0xfffff154	e4 09 01 00
0xfffff158	00 00 00 00
move		

breakpoints

signals

registers

name	value (hex)	value (decimal)	description
r0	0x1	1	
r1	0xfffff294	4294898324	
r2	0x10	16	
r3	0x100	256	
r4	0x10ebc	69308	
r5	0x10158	65880	
r6	0x8b408	570376	
r7	0x10158	65880	
r8	0x0	0	register 8 (64-bit)

STR r3, [r11, #-16] : r3에 저장된 값(256)을 r11-16주소의 메모리에 저장한다. 기존 r11-16는 b의 주솟값이므로 b의 값이 256으로 바뀐 것을 알 수 있다.

정리 : 어셈블리에서 포인터를 사용하여 값을 저장하거나 직접 변수에 값을 저장하는 방식은 같은 변수의 주솟값을 사용하여 값을 수정하기 때문에 결과는 동일하다. 하지만 과정에서 약간의 차이가 있다.

변수로 직접 값을 수정하는 경우 레지스터에 값을 임시저장해둔 후, 변수의 주솟값을 직접 찾아가 수정하는 방식이었다.

포인터를 사용하여 값을 수정하는 경우 먼저 변수의 주솟값을 레지스터에 저장한다. 이후, 다른 레지스터에 변경할 값을 저장하고, 변수의 주솟값이 저장된 레지스터를 통해 메모리에 접근하여 변수의 값을 수정하게 된다.

2) 실습 B

변경된 코드:

```
void example2()
{
    int mem[4] = {0, }; // 16byte array
    char *a;
    int i;

    a = (char*)mem; // Typecasting array's starting address to char-type

    for(i = 0; i < 16; i++)
    {
        if(i % 3 == 0) // The multiple of 3 of the variable i enters this statement
            *(a+i) = 10; // *(a+i) becomes 0x0a
    }
}
```

결과화면:

memory		
0xfffff134	0xfffff153	8
address	hex	
more		
0xfffff134	0a 00 00 0a 00 00 0a 00	
0xfffff13c	00 0a 00 00 0a 00 00 0a	

3) 실습 C

```

1: .text
2: .global example3
3: .type example3, STT_FUNC
4:
5: example3:
6:     sub    sp, sp, #12
7:     str    r3, [sp, #0]
8:     str    r4, [sp, #4]
9:     str    r5, [sp, #8]
10:
11:     sub    sp, sp, #16
12:     mov    r3, #1
13:     str    r3, [sp, #0]
14:
15:     mov    r4, #2
16:     str    r4, [sp], #4
17:
18:     mov    r4, #0
19:     str    r4, [sp]
20:
21:     mov    r5, #3
22:     str    r5, [sp, #0]!
23:
24:     mov    r3, #4
25:     mov    r0, #-1
26:     str    r3, [sp, r0, lsl #2]
27:

```

address	hex	char
0xfffff134	01 00 00 00
0xfffff138	14 0f 01 00
0xfffff13c	58 01 01 00	X...
0xfffff140	08 b4 00 00
0xfffff144	7c 05 01 00
0xfffff148	14 0f 01 00
0xfffff14c	58 01 01 00	X...
0xfffff150	00 00 00 00

13번째 줄 **STR r3, [sp, #0]** : Pre-indexing 사용, offset을 먼저 계산하여 r3의 값(1)을 $sp+0(0xfffff134)$ 에 해당하는 주소에 저장한다.

sp	0xfffff134	4294897972
lr	0x18530	66864
pc	0x1079c	67484

현재 sp값은 0xfffff134이다. 위 코드 수행 후 변화는 없다.

```

1: .text
2: .global example3
3: .type example3, STT_FUNC
4:
5: example3:
6:     sub    sp, sp, #12
7:     str    r3, [sp, #0]
8:     str    r4, [sp, #4]
9:     str    r5, [sp, #8]
10:
11:     sub    sp, sp, #16
12:     mov    r3, #1
13:     str    r3, [sp, #0]
14:
15:     mov    r4, #2
16:     str    r4, [sp], #4
17:
18:     mov    r4, #0
19:     str    r4, [sp]
20:
21:     mov    r5, #3
22:     str    r5, [sp, #0]!
23:
24:     mov    r3, #4
25:     mov    r0, #-1
26:     str    r3, [sp, r0, lsl #2]
27:

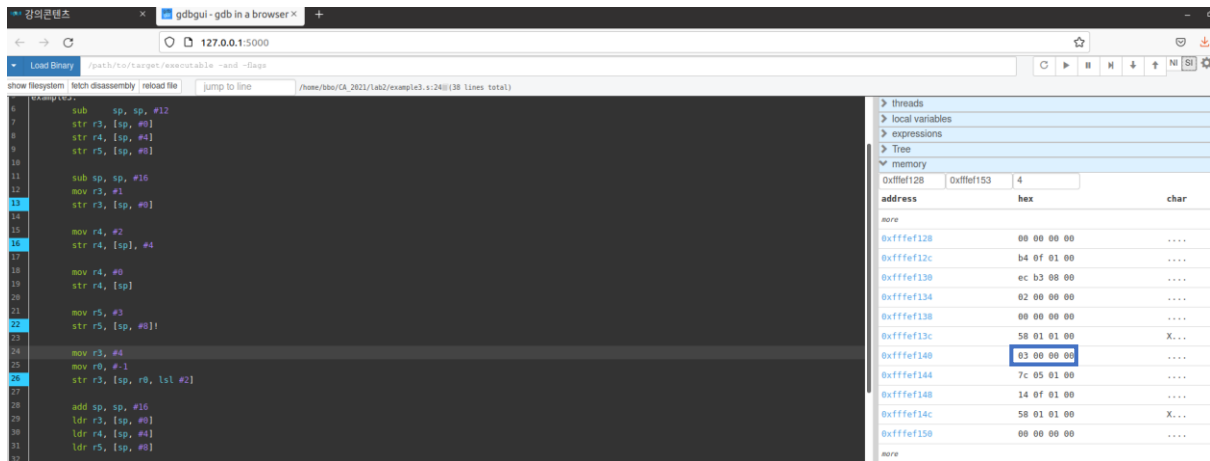
```

address	hex	char
0xfffff128	00 00 00 00
0xfffff12c	b4 0f 01 00
0xfffff130	ec b3 00 00
0xfffff134	02 00 00 00
0xfffff138	14 0f 01 00
0xfffff13c	58 01 01 00	X...
0xfffff140	08 b4 00 00
0xfffff144	7c 05 01 00
0xfffff148	14 0f 01 00
0xfffff14c	58 01 01 00	X...
0xfffff150	00 00 00 00

16번째 줄 **STR r4, [sp], #4** : Post-indexing 사용, r4의 값(2)를 sp가 가리키는 주소(0xfffff134)에 저장한다. 이후 sp의 값을 4만큼 더한 값으로 업데이트한다.

sp	0xfffff138	4294897976
lr	0x18530	66864
pc	0x107a4	67492

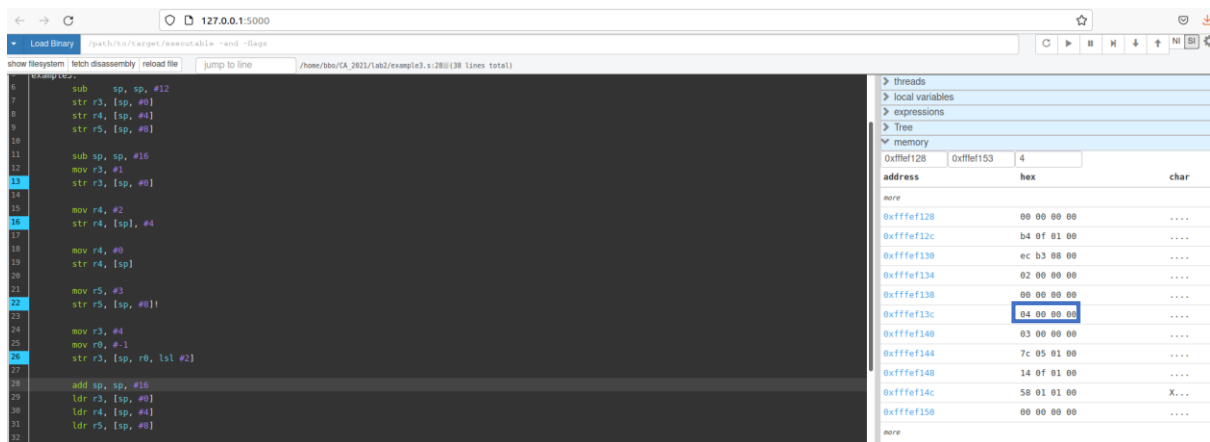
이제 sp의 값은 초기 sp 값에서 +4된 상태다.



22번째 줄 STR r5, [sp, #8]! : Auto-indexing 사용, offset을 먼저 계산하여 r5의 값(8)을 sp+8에 해당하는 주소에 저장한다. 하지만 초기 sp값에서 +4된 상태이므로, 초기 sp값과 비교하면 r5의 값(8)은 sp+12의 위치(0xfffff140)에 저장된다. 이후 sp의 값을 8만큼 더한 값으로 업데이트 한다.

sp	0xfffff140	4294897984
lr	0x10530	66864
pc	0x107b4	67508

이제 sp의 값은 초기 sp의 값에서 +12된 상태다.

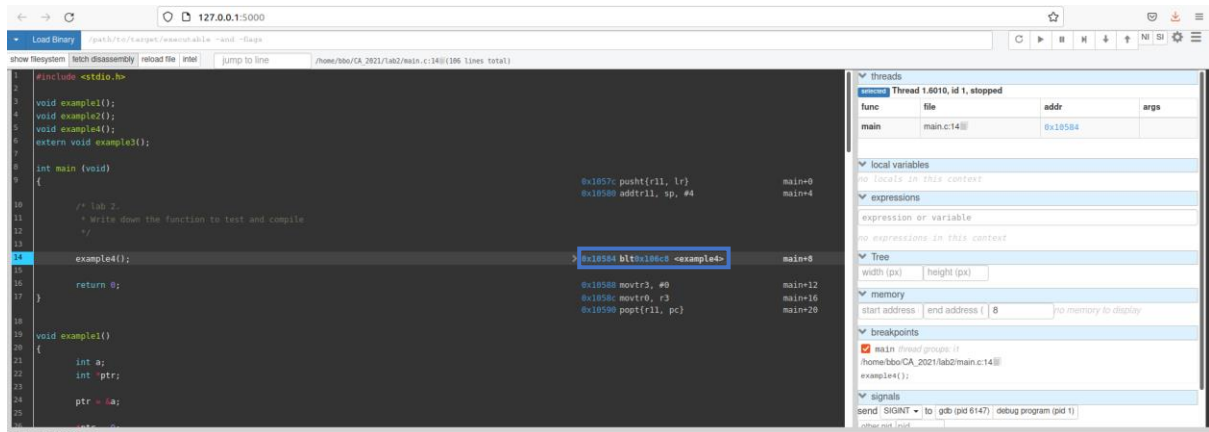


226 줄 STR r3, [sp, r0, lsl #2] : Post-indexing 사용, offset을 먼저 계산하게 된다. r0의 값(-1)을 left shift 2만큼 연산하면 결과값은 -4다. 따라서 r3의 값(4)를 sp-4의 주소에 저장한다. 하지만 초기 sp값에서 +12된 상태이므로, 초기 sp값과 비교하면 r3의 값(4)는 sp+8의 위치(0xfffff13c)에 저장된다.

sp	0xfffff140	4294897984
lr	0x10530	66864
pc	0x107c0	67520

sp 값에는 변화가 없다.

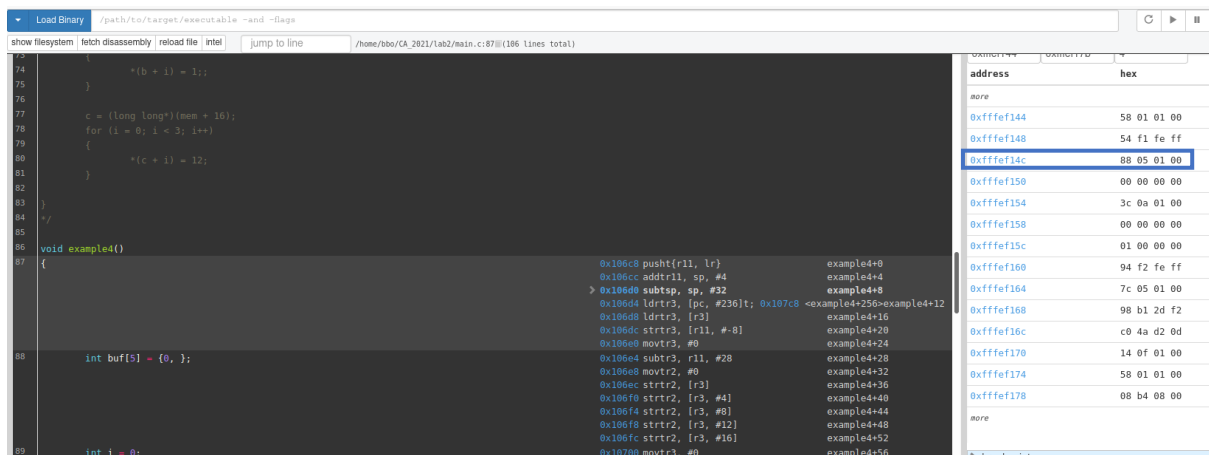
4) 실습 D-1



RETURN을 위해 저장할 PC값 : $0x10584 + 4 = 0x10588$

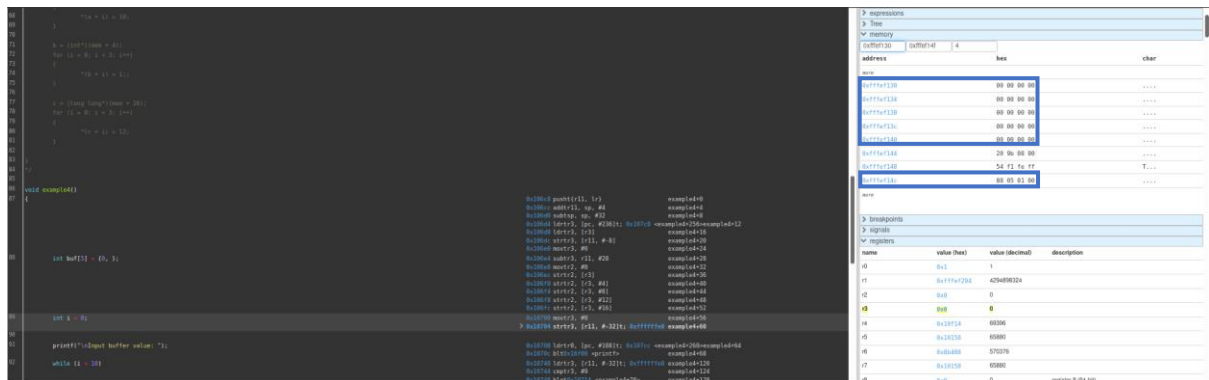
lr	0x10588	66952
pc	0x106c8	67272

branch시 lr에 위의 PC값을 저장, pc는 example4로 이동

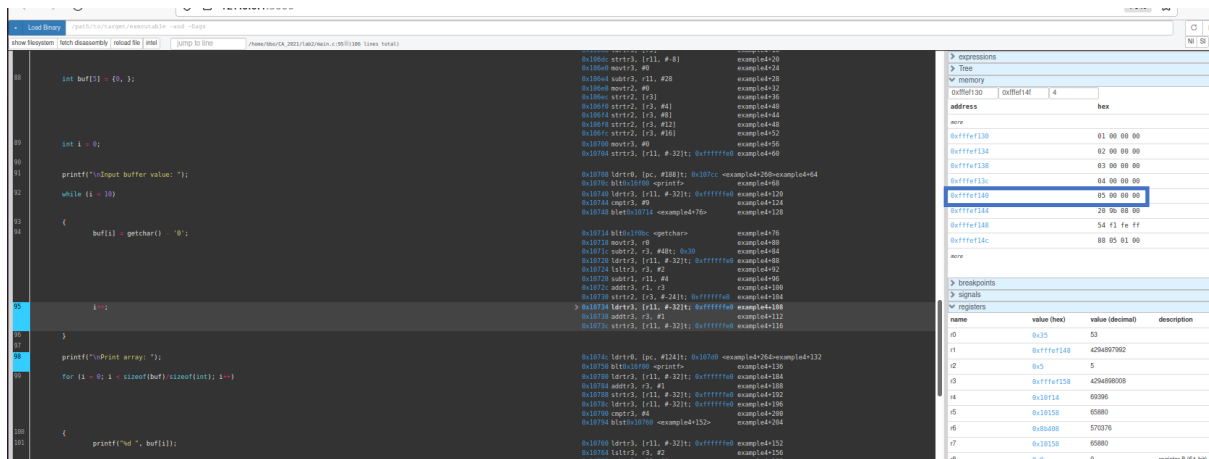


lr을 스택에 push한 주소값 : 0xffff14c

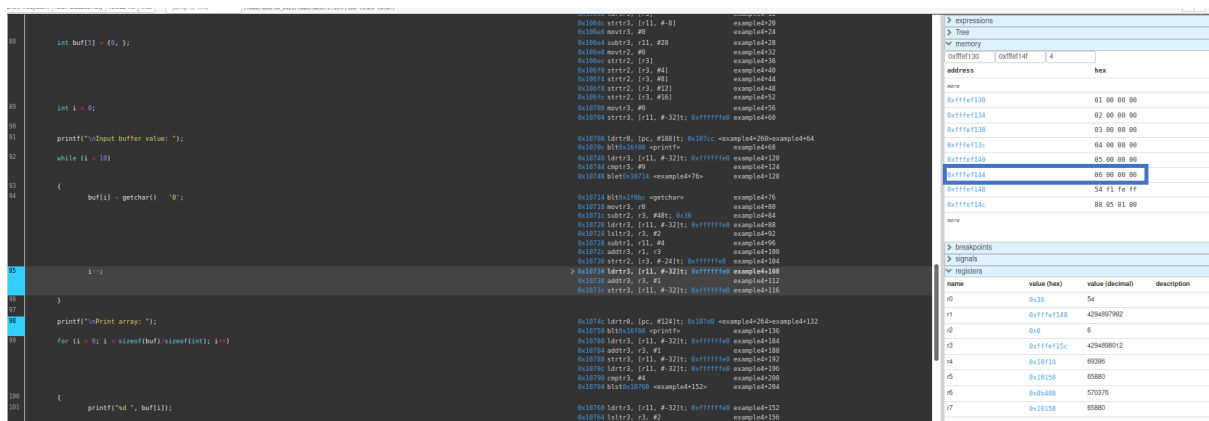
5) 실습 D-2



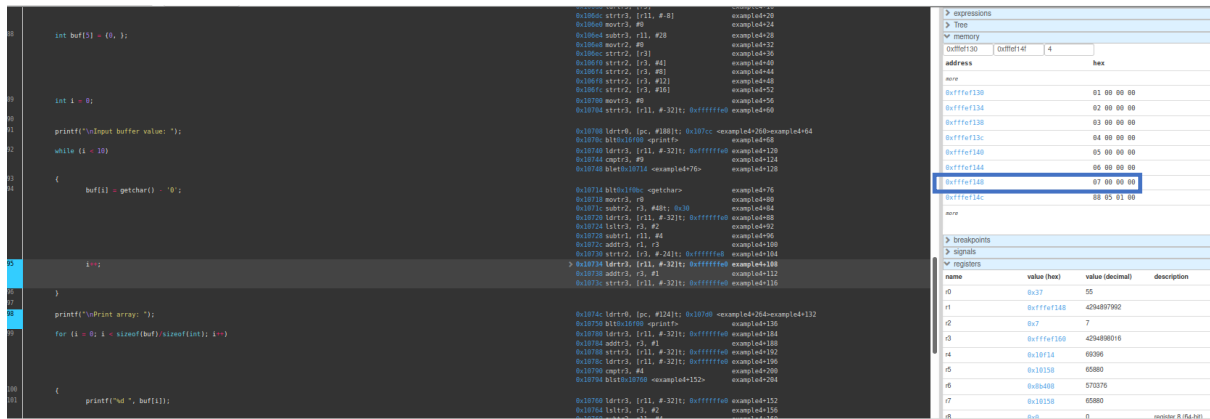
반복문 실행 이전 메모리의 상태를 보면 0xfffff130~0xfffff140까지 배열buf의 공간이고, 0xfffff14c에 return해야 할 주소가 있음을 확인할 수 있다. Input으로는 "1234567891"을 주었다.



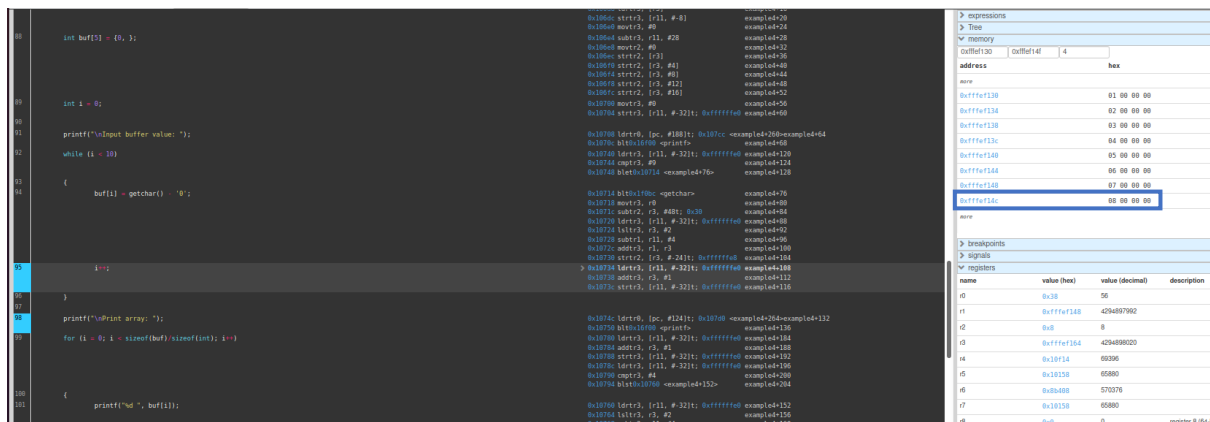
반복문을 5번 실행하여 0xfffff130에서부터 차례로 1,2,3,4,5의 값이 저장되었다. 이제 버퍼가 사용하는 메모리 범위는 모두 사용하였다. 이제부터 배열에 정해진 메모리를 넘어 다른 값들을 변경하게 된다.



반복문을 1회 더 진행시켰다. 0xfffff144에 저장된 0x00008b20의 값을 6으로 변경했다.

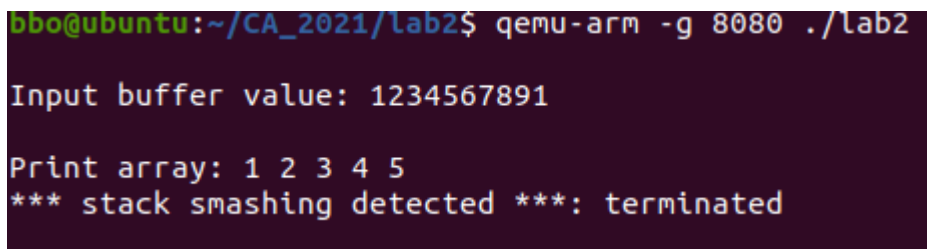


반복문을 1회 더 진행시켰다. 0xfffff148에 저장된 0xfffff154의 값을 7로 변경했다.



반복문을 1회 더 진행시켰다. 0xfffff14c에 저장된 0x00010588의 값을 8로 변경했다.

0xfffff14c는 return해야할 pc주소가 저장된 메모리공간이다. 이를 8로 변경하여 메모리가 깨진 상황이다.



example4함수가 끝나고 return할 때 에러가 나오는 것을 확인할 수 있다.

6) 실습 D-3

```

70     b = (int*)calloc(4);
71     for (i = 0; i < 3; i++)
72     {
73         *(b + i) = 21;
74     }
75
76     c = (long long*)calloc(20);
77     for (i = 0; i < 20; i++)
78     {
79         *(c + i) = 32;
80     }
81 }
82
83 //
84
85 void example()
86 {
87     int buf[5] = {0, };
88     int i = 0;
89
90     printf("Input buffer value: ");
91     while (i < 5)
92     {
93         buf[i] = getch() - '0';
94         i++;
95     }
96
97     printf("Input array: ");
98     for (i = 0; i < sizeof(buf)/sizeof(int); i++)
99     {
100         printf("%d ", buf[i]);
101     }
102
103     printf("\n");
104 }
105
106 //end of file

```

i를 5로 수정한 후 코드의 모습. i가 5로 buf가 가지고 있는 메모리공간에 딱 맞게 Input을 저장하게 된다. 따라서 buf에 Input을 채워넣는 반복문이 끝나고도 0xffff14c에 저장되어있는 return 주소 0x10588이 저장되어 있는 것을 확인할 수 있다.

[illegible]

레지스터 11번 r11이 return시 필요한 pc주소를 저장하고 있는 메모리의 주소를 저장하고 있는 것을 볼 수 있다. 이로써 실습 D-2와 달리 정상 수행이 가능하다.

```
bbo@ubuntu:~/CA_2021/lab2$ qemu-arm -g 8080 ./lab2
Input buffer value: 1234567891
Print array: 1 2 3 4 5
bbo@ubuntu:~/CA_2021/lab2$
```

터미널에서도 에러없이 정상 수행됨을 확인했다.